Search-based Approaches for Local Blackbox Deobfuscation
Understand, Improve and Mitigate

Grégoire Menguy
gregoire.menguy@cea.fr
Université Paris-Saclay, CEA, List
France
Richard Bonichon
richard.bonichon@nomadic-labs.com
Nomadic Labs
France
Sébastien Bardin
sebastien.bardin@cea.fr
Université Paris-Saclay, CEA, List
France
Cauim de Souza de Lima
cauimsouza@gmail.com
Université Paris-Saclay, CEA, List
France

ABSTRACT
Code obfuscation aims at protecting Intellectual Property and other secrets embedded into software from being retrieved. Recent works leverage advances in artificial intelligence (AI) with the hope of getting blackbox deobfuscators completely immune to standard (whitebox) protection mechanisms. While promising, this new field of AI-based, and more specifically search-based blackbox deobfuscation, is still in its infancy. In this article we deepen the state of search-based blackbox deobfuscation in three key directions: understand the current state-of-the-art, improve over it and design dedicated protection mechanisms. In particular, we define a novel generic framework for search-based blackbox deobfuscation encompassing prior work and highlighting key components; we are the first to point out that the search space underlying code deobfuscation is too unstable for simulation-based methods (e.g., Monte Carlo Tree Search used in prior work) and advocate the use of robust methods such as S-metaheuristics; we propose the new optimized search-based blackbox deobfuscator Xyntia which significantly outperforms prior work in terms of success rate (especially with small time budget) while being completely immune to the most recent anti-analysis code obfuscation methods; and finally we propose two novel protections against search-based blackbox deobfuscation, allowing to counter Xyntia powerful attacks.

CCS CONCEPTS
• Computing methodologies → Game tree search; Heuristic function construction; Game tree search; • Security and privacy → Software reverse engineering.

KEYWORDS
Binary-level code analysis, deobfuscation, artificial intelligence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference’17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5062-1/17/07...
https://doi.org/10.1145/3122445.3122456

1 INTRODUCTION
Context. Software contain valuable assets, such as secret algorithms, business logic or cryptographic keys, that attackers may try to retrieve. The so-called Man-At-The-End-Attacks scenario (MATE) considers the case where software users themselves are adversarial and try to extract such information from the code. Code obfuscation [12, 13] aims at protecting codes against such attacks, by transforming a sensitive program into a functionally equivalent program that is more “difficult” (more expensive, for example, in money or time) to understand or modify. On the flip side, code deobfuscation aims to extract information from obfuscated codes.

Whitebox deobfuscation techniques, based on advanced symbolic program analysis, have proven extremely powerful against standard obfuscation schemes [3, 5, 10, 22, 28, 30, 36] – especially in local attack scenarios where the attacker analyses pre-identified parts of the code (e.g., trigger conditions). But they are inherently sensitive to the syntactic complexity of the code under analysis, leading to recent and effective countermeasures [12, 25, 26, 37].

Search-based blackbox deobfuscation. Despite being rarely complete or sound, artificial intelligence (AI) techniques are flexible and often provide good enough solutions to hard problems in reasonable time. They have been therefore recently applied to binary-level code deobfuscation. The pioneering work by Blazytko et al. [7] shows how Monte Carlo Tree Search (MCTS) [9] can be leveraged to solve local deobfuscation tasks by learning the semantics of pieces of protected codes in a blackbox manner, in principle immune to the syntactic complexity of these codes. Their method and prototype, Syntia, have been successfully used to reverse state-of-the-art protectors like VMProtect [34], Themida [27] and Tigrress [11], drawing attention from the software security community [8].

Problem. While promising, search-based blackbox (code) deobfuscation techniques are still not well understood. Several key questions of practical relevance (e.g., deobfuscation correctness and quality, sensitivity to time budget) are not addressed in Blazytko et
Contributions. Our main contributions are the following: application of S-metaheuristics [32] instead of MCTS. new insights for efficient blackbox deobfuscation, and promote the attacks. Especially, we study the underlying search space, bringing mitigate through a careful formalization and exploration of the design space improve blackbox methods, (2) understand elements by Blazytko et al. in order to better understand search-based blackbox deobfuscation in the following three key directions: (1) generalize the initial Syntia proposal and refine the initial experiments by Blazytko et al. in order to better understand search-based blackbox methods, (2) improve the current state-of-the-art (Syntia) through a careful formalization and exploration of the design space and evaluate the approach against greybox and program synthesis methods, and finally (3) study how to mitigate such blackbox attacks. Especially, we study the underlying search space, bringing new insights for efficient blackbox deobfuscation, and promote the application of S-metaheuristics [32] instead of MCTS.

Goal. We focus on advancing the current state of search-based blackbox deobfuscation in the following three key directions: (1) generalize the initial Syntia proposal and refine the initial experiments by Blazytko et al. in order to better understand search-based blackbox methods, (2) improve the current state-of-the-art (Syntia) through a careful formalization and exploration of the design space and evaluate the approach against greybox and program synthesis methods, and finally (3) study how to mitigate such blackbox attacks. Especially, we study the underlying search space, bringing new insights for efficient blackbox deobfuscation, and promote the application of S-metaheuristics [32] instead of MCTS.

Contributions. Our main contributions are the following:

- We refine Blazytko et al.’s experiments in a systematic way, highlighting new strengths and new weaknesses of the initial Syntia proposal for search-based blackbox deobfuscation (Section 4). Especially, Syntia (based on Monte Carlo Tree Search, MCTS) is far less efficient than expected for small time budgets (typical usage scenario) and lacks robustness;
- We propose a missing formalization of blackbox deobfuscation (Section 4) and dig into Syntia internals to rationalize our observations (Section 4.4). It appears that the search space underlying blackbox code deobfuscation is too unstable to rely on MCTS – especially assigning a score to a partial state through simulation leads to poor estimations. As a result, Syntia is here almost enumerative;
- We propose to see blackbox deobfuscation as an optimization problem rather than a single player game (Section 5), allowing to reuse S-metaheuristics [32], known to be more robust than MCTS on unstable search spaces (especially, they do not need to score partial states). We propose Xyntia (Section 5), an search-based blackbox deobfuscator using Iterated Local Search (ILS) [24], known among S-metaheuristics for its robustness. Thorough experiments show that Xyntia keeps the benefits of Syntia while correcting most of its flaws. Especially, Xyntia significantly outperforms Syntia, synthesizing twice more expressions with a budget of 1 s/expr than Syntia with 600 s/expr. Other S-metaheuristics also clearly beat MCTS, even if they are less effective here than ILS;
- We evaluate Xyntia against other state-of-the-art attackers (Section 6), namely the QSynth greybox deobfuscator [16], program synthesizers CVC4 [6] and STOKE [29], and pattern-matching based simplifiers. Xyntia outperforms all of them – it finds 2× more expressions and is 30× faster than QSynth on heavy protections;
- We evaluate Xyntia against state-of-the-art defenses (Section 7), especially recent anti-analysis proposals [14, 25, 31, 35, 37]. As expected, Xyntia is immune to such defenses. In particular, it successfully bypasses side-channels [31], path explosion [25] and MBA [37]. We also use it to synthesize VM-handlers from state-of-the-art virtualizers [11, 34];
- Finally, we propose the two first protections against search-based blackbox deobfuscation (Section 8). We observe that all phases of blackbox techniques can be thwarted (hypothesis, sampling and learning), we propose two practical methods exploiting these limitations and we discuss them in the context of virtualization-based obfuscation: (1) semantically complex handlers; (2) merged handlers with branch-less conditions. Experiments show that both protections are highly effective against blackbox attacks.

We hope that our results will help better understand search-based deobfuscation, and lead to further progress in this promising field.

Availability. Benchmarks and code are available online.1 Additional experimental data will be made available in a separate technical report.

2 BACKGROUND

2.1 Obfuscation

Program obfuscation [12, 13] is a family of methods designed to make reverse engineering (understanding programs internals) hard. It is employed by manufacturers to protect intellectual property and by malware authors to hinder analysis. It transforms a program into a functionally equivalent, more complex program with an acceptable performance penalty. Obfuscation does not ensure that a program cannot be understood – this is impossible in the MATE context – but aims to delay the analysis as much as possible in order to make it unprofitable. Thus, it is especially important to protect from automated deobfuscation analyses (anti-analysis obfuscation). We present here two important obfuscation methods.

Mixed Boolean-Arithmetic (MBA) encoding [37] transforms an arithmetic and/or Boolean expression into an equivalent one, combining arithmetic and Boolean operations. It can be applied iteratively to increase the syntactic complexity of the expression. Eyrolles et al. [18] show that SMT solvers struggle to answer equivalence requests on MBA expressions, preventing the automated simplification of protected expressions by symbolic methods.

Virtualization [35] translates an initial code P into a bytecode B together with a custom virtual machine. Execution of the obfuscated code can be divided in 3 steps (Fig. 1): (1) fetch the next bytecode instruction to execute, (2) decode the bytecode and find the corresponding handler, (3) and finally execute the handler. Virtualization hides the real control-flow graph (CFG) of P, and reversing the handlers is key for reversing the VM. Virtualization is notably used in malware [19, 33].

![Figure 1: Virtualization based obfuscation](https://tinyurl.com/y34dcaus)
2.2 Deobfuscation

Deobfuscation aims at reverting an obfuscated program back to a form close enough to the original one, or at least to a more understandable version. Along the previous years, symbolic deobfuscation methods based on advanced program analysis techniques have proven to be very efficient at breaking standard protections [3, 5, 10, 22, 28, 30, 36]. However, very effective countermeasures start to emerge, based on deep limitations of the underlying code-level reasoning mechanisms and potentially strongly limiting their usage [3, 25, 26, 31, 35]. Especially, all such methods are ultimately sensitive to the syntactic complexity of the code under analysis.

2.3 Search-based blackbox deobfuscation

Search-based blackbox deobfuscation has been recently proposed by Blazytko et al. [7], implemented in the Syntia tool, to learn the semantics of well-delimited code fragments, e.g. MBA expressions or VM handlers. The code under analysis is seen as a blackbox that can only be queried (i.e., executed under chosen inputs to observe results). Syntia samples input-output (I/O) relations, then uses a learning engine to find an expression mapping sampled inputs to their observed outputs. Because it relies on a limited number of samples, results are not guaranteed to be correct. However, being fully blackbox, it is in principle insensitive to syntactic complexity.

Scope. Syntia tries to infer a simple semantics of heavily obfuscated local code fragments – e.g., trigger based conditions or VM handlers. Understanding these fragments is critical to fulfill analysis.

Workflow. Syntia workflow is representative of search-based blackbox deobfuscators. First, it needs (1) a reverse window i.e., a subset of code to work on; (2) the location of its inputs and outputs. Consider the code in Listing 1 evaluating a condition at line 4. To understand this condition, a reverser focuses on the code between lines 1 and 3. This code segment is our reverse window. The reverser then needs to locate relevant inputs and outputs. The condition at line 4 is performed on t3. This is our output. The set of inputs contains any variables (registers or memory locations at assembly level) influencing the outputs. Here, inputs are x and y. Armed with these information, Syntia samples inputs randomly and observes resulting outputs. In our example, it might consider the samples (x → 1, y → 2), (x → 0, y → 1) and (x → 3, y → 4) which respectively evaluate t3 to 3, 1 and 7. Syntia then synthesizes an expression matching these observed behaviors, using Monte Carlo Tree Search (MCTS) over the space of all possible (partial) expressions. Here, it rightly infers that t3 ← x + y and the reverser concludes that the condition is x + y = 5, where a symbolic method will typically simply retrieve that (x ∨ 2y) × 2 − (x ⊕ 2y) − y = 5.

Listing 1: Obfuscated condition

```c
int t1 = 2 * y;
int t2 = x | t1;
int t3 = t2 * 2 - (x ^ t1) - y;
if (t3 == 5) ...
```

3 MOTIVATION

3.1 Attacker model

In the MATE scenario, the attacker is the software user himself. He has only access to the obfuscated version of the code under analysis and can read or run it at will. We consider that the attacker is highly skilled in reverse engineering but has limited resources in terms of time or money. We see reverse engineering as a human-in-the-loop process where the attacker combines manual analysis with automated state-of-the-art deobfuscation methods (slicing, symbolic execution, etc.) on critical, heavily obfuscated code fragments like VM handlers or trigger-based conditions. Thus, an effective defense strategy is to thwart automated deobfuscation methods.

3.2 Syntactic and semantic complexity

We now intuitively motivate the use of blackbox deobfuscation. Consider that we reverse a piece of software protected through virtualization. We need to extract the semantics of all handlers, which usually perform basic operations like h(x, y) = x + y. Understanding h is trivial, but it can be protected to hinder analysis. Eq. (1) shows how MBA encoding hides h semantics.

\[
h(x, y) = x + y \xrightarrow{\text{MBA}} (x \lor 2y) \times 2 - (x \land 2y) - y \quad (1)
\]

Such encoding syntactically transforms the expression to make it incomprehensible while preserving its semantics. To highlight the difference between syntax and semantics, we distinguish:

1. The syntactic complexity of expression e is the size of e, i.e. the number of operators used in it;
2. The semantic complexity of expression e is the smallest size of expressions e’ (in a given language) equivalent to e.

For example, in the MBA language, x + y is syntactically simpler than (x ∨ 2y) × 2 − (x ⊕ 2y) − y, yet they have the same semantic complexity as they are equivalent. Conversely, x + y is more semantically complex than (x + y) ∧ 0, which equals 0. We do not claim to give a definitive definition of semantic and syntactic complexity – as smaller is not always simpler – but introduce the idea that two kinds of complexity exist and are independent.

The encoding in Eq. (1) is simple, but it can be repeatedly applied to create a more syntactically complex expression, leading the reverser to either give up or try to simplify it automatically. White-box methods based on symbolic execution (SE) [28, 36] and formula simplifications (in the vein of compiler optimizations) can extract the semantics of an expression, yet they are sensitive to syntactic complexity and will not return simple versions of highly obfuscated expressions. Conversely, blackbox deobfuscation treats the code as a blackbox, considering only sampled I/O behaviors. Thus increasing syntactic complexity, as usual state-of-the-art protections do, has simply no impact on blackbox methods.

3.3 Blackbox deobfuscation in practice

We now present how blackbox methods integrate in a global deobfuscation process and highlight crucial properties they must hold.

Global workflow. Reverse engineering can be fully automated, or handmade by a reverser, leveraging tools to automate specific tasks. While the deobfuscation process operates on the whole obfuscated binary, blackbox modules can be used to analyze parts of the code like conditions or VM handlers. Upon meeting a complex code fragment, the blackbox deobfuscator is called to retrieve a simple semantic expression. After synthesis succeeds, the inferred expression is used to help continue the analysis.
We propose a general view of search-based code deobfuscation fitting state-of-the-art solutions [7, 16]. We also extend the evaluation of Syntia by Blazytko et al. [7], highlighting both some previously unreported weaknesses and strengths. From that we derive general lessons on the (in)adequacy of MCTS for code deobfuscation, that will guide our new approach (Section 5).

4 UNDERSTAND BLACKBOX DEOBFUSCATION

We propose a general view of search-based code deobfuscation fitting state-of-the-art solutions [7, 16]. We also extend the evaluation of Syntia by Blazytko et al. [7], highlighting both some previously unreported weaknesses and strengths. From that we derive general lessons on the (in)adequacy of MCTS for code deobfuscation, that will guide our new approach (Section 5).

4.1 Problem at hand

Search-based deobfuscation takes an obfuscated expression and tries to infer an equivalent one with lower syntactic complexity. Such problem can be stated as following:

Deobfuscation. Let \( e \) and \( obf \) be 2 equivalent expressions such that \( obf \) is an obfuscated version of \( e \) – note that \( obf \) is possibly much larger than \( e \). Deobfuscation aims to infer an expression \( e' \) equivalent to \( obf \) (and \( e \)), but with size similar to \( e \). Such problem can be approached in three ways depending on the amount of information given to the analyzer:

- **Blackbox** We can only run \( obf \). The search is thus driven by sampled I/O behaviors. Syntia [7] is a blackbox approach;
- **Greybox** Here \( obf \) is executable and readable but the semantics of its operators is mostly unknown. The search is driven by previously sampled I/O behaviors which can be applied to subparts of \( obf \). QSynth [16] is a greybox solution;
- **Whitebox** The analyzer has full access to \( obf \) (run, read) and the semantics of its operators is precisely known. Thus, the search can profit from advanced pattern matching and symbolic strategies. Standard static analysis falls in this category.

**Blackbox methods.** Search-based blackbox deobfuscators follow the framework given in Algorithm 1. In order to deobfuscate code, one must detail a sampling strategy (i.e., how inputs are generated), a learning strategy (i.e., how to learn an expression mapping sampled inputs to observed outputs) and a simplification postprocess. For example, Syntia samples inputs randomly, uses Monte Carlo Tree Search (MCTS) [9] as learning strategy and leverages the Z3 SMT solver [17] for simplification. The choice of the sampling and learning strategies is critical. For example, too few samples could lead to incorrect results while too many could impact the search efficiency, and an inappropriate learning algorithm could impact robustness or speed.

Let us now turn to discussing Syntia learning strategy. We show that using MCTS leads to disappointing performances and give insights to understand why.

**Algorithm 1 Search-based blackbox deobfuscation framework**

**Inputs:**
- Code: code to analyze
- Sample: sampling strategy
- Learn: learning strategy
- Simplify: expression simplifier

**Output:** learned expression or Failure

1. procedure DEOBFUSCATE(Code, Sample, Learn)
2. Oracle ← Sample(Code)
3. succ, expr ← Learn(Oracle)
4. if succ = True then return Simplify(expr)
5. else return Failure

4.2 Evaluation of Syntia

We extend Syntia evaluation and tackle the following questions left unaddressed by Blazytko et al. [7].

**RQ1** Are results stable across different runs?

This is desirable due to the stochastic nature of MCTS;

**RQ2** Is Syntia fast, robust and does it infer simple and correct results?

Syntia offers a priori no guarantee of correctness nor quality. Also, we consider small time budget (1s), adapted to human-in-the-loop scenarios but absent from the initial evaluation;

**RQ3** How is synthesis impacted by the set of operators size?

Syntia learns expressions over a search space fixed by predefined grammars. Intuitively, the more operators in the grammar, the harder it will be to converge to a solution. We use 3 sets of operators to assess this impact.

4.2.1 Experimental setup. We distinguish the success rate (number of expressions inferred) from the equivalence rate (number of expressions inferred and equivalent to the original one). The equivalence rate relies on the Z3 SMT solver [17] with a timeout of 10s. Since Z3 timeouts are inconclusive answers, we define a notion of equivalence range: its lower bound is the proven equivalence rate (number of expressions proven to be equivalent) while its upper bound is the optimistic equivalence rate (expressions not proven different, i.e., optimistic = proven + #timeout). The equivalence rate is within the equivalence range, while the success rate is higher than the optimistic equivalence rate. Finally, we define the quality of an expression as the ratio between the number of operators in recovered and target expressions. It estimates the syntactic complexity of inferred expressions compared to the original ones. A quality of 1 indicates a perfect result: the recovered expression has the same size as the target expression.

**Benchmarks.** We consider two benchmark suites: B1 and B2. B1 comes from Blazytko et al. [7] and was used to evaluate Syntia. It comprises 500 randomly generated expressions with up to 3

[https://github.com/RUB-SysSec/syntia/tree/master/samples/mba/tigress](https://github.com/RUB-SysSec/syntia/tree/master/samples/mba/tigress)
arguments, and simple semantics. It aims at representing state-of-the-art VM-based obfuscators. However, we found that B1 suffers from several significant issues: (1) it is not well distributed over the number of inputs and expression types, making it unsuitable for fine-grained analysis; (2) only 216 expressions are unique modulo renaming – the other 284 expressions are \( \alpha \)-equivalent, like \( x \cdot y \) and \( a \cdot b \). These problems threaten the validity of the evaluation.

We thus propose a new benchmark B2 consisting of 1,110 randomly generated expressions, better distributed according to the number of inputs and the nature of operators – see Table 1. We use three categories of expressions: Boolean, Arithmetic and Mixed Boolean-Arithmetic, with 2 to 6 inputs. Especially, expressions are spread equally between categories to prevent biased results. Each expression has an Abstract Syntax Tree (AST) of maximal height 3. As a result, B2 is more challenging than B1 and enables a finer-grained evaluation. Considering such diverse and complex expressions is crucial as blackbox obfuscation evolves in an adversarial context where limitations can be exploited to thwart analysis.

Note that we also consider QSynth datasets [16] in Section 6, developed by the Quarkslab R&D company.

### Operator sets

Table 2 introduces three operator sets: Full, Expr and MBA. We use these to evaluate sensitivity to the search space and answer RQ3. Expr is as expressive as Full even if Expr \( \subset \) Full. MBA can only express Mixed Boolean-Arithmetic expressions [37].

<table>
<thead>
<tr>
<th>Type</th>
<th># Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>Arith</td>
</tr>
<tr>
<td>#Expr</td>
<td></td>
</tr>
<tr>
<td>370</td>
<td>370</td>
</tr>
<tr>
<td>150</td>
<td>600</td>
</tr>
<tr>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 1: Distribution of samples in benchmark B2

### Conclusion

Syntia is stable and returns correct and simple results. Yet, it is not efficient enough (solves only few expressions on B2, heavily impacted by time budget) and not robust (number of inputs and expression type).

### Table 3: Syntia depending on the timeout per expression (B2)

<table>
<thead>
<tr>
<th></th>
<th>1s</th>
<th>10s</th>
<th>60s</th>
<th>600s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ. Rate</td>
<td>16.5%</td>
<td>25.6%</td>
<td>34.5%</td>
<td>42.3%</td>
</tr>
<tr>
<td>Equiv. Range</td>
<td>16.3%</td>
<td>25.1 - 25.3%</td>
<td>33.7 - 34.0%</td>
<td>41.4 - 41.6%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.35</td>
<td>0.49</td>
<td>0.59</td>
<td>0.67</td>
</tr>
</tbody>
</table>

RQ3. Default Syntia synthesizes expressions over the Full set of operators. To evaluate its sensitivity to the search space we run it over Full, Expr and MBA. Smaller sets do exhibit higher success rates (42% on MBA) but results remain disappointing. Syntia is sensitive to the size of the operator set but is inefficient even with MBA.

### 4.3 Optimal Syntia

To ensure the conclusions given in Section 4.4 apply to MCTS and not only to Syntia, we study Syntia extensively to find better set ups for the following parameters: simulation depth, SA-UCT value (configuring the balance between exploitative and explorative behaviors), number of I/O samples and distance. Optimizing Syntia parameters slightly improves its results which stay disappointing (at best, \( \approx 50\% \) of success rate on MBA in 60 s/expr.).

### Conclusion

Syntia is sensitive to the search space and answer RQ3. Expr is as expressive as Full even if Expr \( \subset \) Full. MBA can only express Mixed Boolean-Arithmetic expressions [37].

<table>
<thead>
<tr>
<th></th>
<th>1s</th>
<th>10s</th>
<th>60s</th>
<th>600s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ. Rate</td>
<td>16.5%</td>
<td>25.6%</td>
<td>34.5%</td>
<td>42.3%</td>
</tr>
<tr>
<td>Equiv. Range</td>
<td>16.3%</td>
<td>25.1 - 25.3%</td>
<td>33.7 - 34.0%</td>
<td>41.4 - 41.6%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.35</td>
<td>0.49</td>
<td>0.59</td>
<td>0.67</td>
</tr>
</tbody>
</table>

RQ3. Default Syntia synthesizes expressions over the Full set of operators. To evaluate its sensitivity to the search space we run it over Full, Expr and MBA. Smaller sets do exhibit higher success rates (42% on MBA) but results remain disappointing. Syntia is sensitive to the size of the operator set but is efficient even with MBA.

### Conclusion

Syntia is stable, correct and returns simple results. Yet, it is heavily impacted by the time budget and lacks robustness. It thus fails to meet the requirements given in Section 3.3.

### 4.4 MCTS for deobfuscation

Let us explore whether these issues are related to MCTS.

#### Monte Carlo Tree Search

MCTS creates here a search tree where each node is an expression which can be terminal (e.g. \( a + 1 \), where \( a \) is a variable) or partial (e.g. \( U + a \), where \( U \) is a non-terminal symbol). The goal of MCTS is to expand the search tree smartly, focusing on most pertinent nodes first. Evaluating the pertinence of a terminal node is done by sampling (computing here a distance between the evaluation of sampled inputs over the node expression against their expected output values). For partial nodes, MCTS relies on simulation: random rules of the grammar are applied to the expression (e.g., \( U + a \rightarrow b + a \)) until it becomes terminal and is evaluated. As an example, let \( (a \mapsto 1, b \mapsto 0), (a \mapsto 0, b \mapsto 1) \) be the sampled inputs. The expression \( b + a \) (simulated from \( U + a \)) evaluates them to \( (1,1) \). If the ground-truth outputs are 1 and -1, the distance will equal \( \delta(1,1) + \delta(1,1) \) where \( \delta \) is a chosen distance function. We call the result the pertinence measure. The closer it is to 0, the more pertinent the node \( U + a \) is considered and the more the search will focus on it.
Analysis. This simulation-based pertinence estimation is not reliable in our code deobfuscation setting.

- We present in Fig. 2, for different non-terminal nodes, the distance values computed through simulations. We observe that from a starting node, a random simulation can return drastically different results. It shows that the search space is very unstable and that relying on simulation is misleading (especially in our context where time budget is small);
- Moreover, our experiments show that in practice Syntia is not guided by simulations and behaves almost as if it were an enumerative (BFS) search – MCTS where simulations are non informative. As an example, Fig. 3 compares how the distance evolves over time for Syntia and a custom, fully enumerative, MCTS synthesizer: both are very similar. Actually, Syntia and enumerative MCTS perform similarly over B2: with a 60s (resp. 600s) timeout, enumerative MCTS reaches 41.4% (resp. 51.6%) success rate vs. 42.6% (resp. 54.9%) for Syntia (Mba operators set);
- Finally, on B2 (resp. B1) with a timeout of 60s, only 34/341 (resp. 20/376) successfully synthesized expressions are the children of previously most promising nodes. It shows that Syntia successfully synthesized expressions due to its exploratory (i.e., enumerative) behavior rather than to the selection of nodes according to their pertinence.

Conclusion. The search space from blackbox code deobfuscation is too unstable, making MCTS simulations unreliable. MCTS in that setting is then almost enumerative and inefficient. That is why Syntia is slow and not robust, but returns simple expressions.

4.5 Conclusion

While Syntia returns simple results, it only synthesizes semantically simple expressions and is slow. These unsatisfactory results can be explained by the fact that the search space is too unstable, making the use of MCTS unsuitable. In the next section, we show that methods avoiding the manipulation of partial expressions (and thus free from simulation) are better suited to deobfuscation.

5 IMPROVE BLACKBOX DEOBFUSCATION

We define a new search-based blackbox deobfuscator, dubbed Xyn-tia, leveraging S-metaheuristics [32] and Iterated Local Search (ILS) [24] and compare its design to rival deobfuscators. Unlike MCTS, S-metaheuristics only manipulate terminal expressions and do not create tree searches, thus we expect them to be better suited than MCTS for code deobfuscation. Among S-metaheuristics, ILS is particularly designed for unstable search spaces, with the ability to remember the last best solution encountered and to restart the search from that point. We show that these methods are well-guided by the distance function and significantly outperform MCTS in the context of blackbox code deobfuscation.

5.1 Deobfuscation as Optimization

As presented in Section 4, Syntia frames deobfuscation as a single player game. We instead propose to frame it as an optimization problem using ILS as learning strategy.

Blackbox deobfuscation: an optimization problem. Blackbox deobfuscation synthesizes an expression from inputs-outputs samples and can be modeled as an optimization problem. The objective function, noted $f$, measures the similarity between current and ground truth behaviors by computing the sum of the distances between found and objective outputs. The goal is to infer an expression minimizing the objective function over the I/O samples. If the underlying grammar is expressive enough, a minimum exists and matches all sampled inputs to objective outputs, zeroing $f$. The reliability of the found solution depends on the number of I/O samples considered. Too few samples would not restrain search enough and lead to flawed results.

Solving through search heuristics. S-metaheuristics [32] can be advantageously used to solve such optimization problems. A wide range of heuristics exists (Hill Climbing, Random Walk, Simulated Annealing, etc.). They all iteratively improve a candidate solution by testing its “neighbors” and moving along the search space. Because solution improvement is evaluated by the objective function, it is said to guide the search.

Iterated Local Search. Some S-metaheuristics are prone to being stuck in local optima so that the result depends on the initial input chosen. Iterated Local Search (ILS) [24] tackles the problem through iteration of search and the ability to restart from previously seen best solutions. Note that ILS is parameterized by another search heuristics (for us: Hill Climbing). Once a local optimum is
found by this side search, ILS perturbs it and uses the perturbed solution as initial state for the side search. At each iteration, ILS also saves the best solution found. Unlike most other S-metaheuristics (Hill Climbing, Random Walk, Metropolis Hasting and Simulated Annealing, etc.), if the search follows a misleading path, ILS can restore the best seen solution so far to restart from an healthy state.

5.2 Xyntia internals

Xyntia is built upon 3 components: the optimization problem we aim to solve, the oracle which extracts the sampling information from the protected code under analysis and the search heuristics.

Oracle. The oracle is defined by the sampling strategy which depicts how the protected program must be sampled and how many samples are considered. As default, we consider that our oracle samples 100 inputs over the range $[-50; 49]$. Five are not randomly generated but equal interesting constant vectors $(0, 1, -1, \min, \max)$. These choices arise from a systematic study of the different settings to find the best design (see Section 5.4).

Optimization problem. The optimization problem is defined as follow. The search space is the set of expressions expressible using the Expr set of operators (see Table 2), and considers a unique constant value 1. This grammar enables Xyntia to reach optimal results while being as expressive as Syntia [7]. Besides, we consider the objective function:

$$f(\tilde{\sigma}) = \sum_i \log_2(1 + |\sigma_i - \tilde{\sigma}_i|)$$

It computes the Log-arithmetic distance between synthesized expressions outputs ($\tilde{\sigma}$) and sampled ones ($\tilde{\sigma}_i$). The choice of the grammar and of the objective function are respectively discussed in Sections 5.3 and 5.4.

Search. Xyntia leverages Iterated Local Search (ILS) to minimize our objective function and so to synthesize target expressions. We present now how Xyntia is adapted to our context. ILS applies two steps starting from a random terminal (constant value or variable):

- ILS reuses the best expression found so far to perturb it by randomly selecting a node of the AST and replacing it by a random terminal node. The resulting AST is kept even if the distance increases and passed to the next step.
- Iterative Random Mutations: the side search (in our case Hill Climbing) iteratively mutates the input expression until it cannot improve anymore. We estimate that no more improvement can be done after 100 inconclusive mutations. A mutation consists in replacing a randomly chosen node of the abstract syntax tree (AST) by a leaf or an AST of depth one (only one operator) - e.g. $1 + (-a) \rightarrow (-b) + (-a)$. At each mutation, it keeps the version of the AST minimizing the distance function. During mutations, the best solution so far is updated to be restored in the perturbation step. If a solution nullifies the objective function, it is directly returned.

These two operations are iteratively performed until time is out (by default 60s) or an expression mapping all I/O samples is found. Furthermore, as Syntia applies Z3 simplifier to “clean up” recovered expressions, we add a custom post-process expression simplifier, applying simple rewrite rules until a fixpoint is reached.

It significantly improves the quality of the expressions while adding no significant overhead (+2.6ms on average). Xyntia is implemented in OCaml [23], within the BINSEC framework for binary-level program analysis [15]. It comprises $\approx 9k$ lines of code.

5.3 Xyntia evaluation

We now evaluate Xyntia in depth and compare it to Syntia. As with Syntia we answer the following questions:

**RQ4** Are results stable across different runs?

**RQ5** Is Xyntia robust, fast and does it infer simple and correct results?

**RQ6** How is synthesis impacted by the set of operators size?

Configuration. For all our experiments, we default to locally optimal Xyntia (Xyntia$_{opt}$) presented in Section 5.2. It learns expressions over Expr, samples 100 inputs (95 randomly and 5 constant vectors) and uses the Log-arithmetic distance as objective function.

Interestingly, all results reported here also hold (to a lesser extend regarding efficiency) for other Xyntia configurations (Section 5.4), especially these versions consistently beat Syntia.

**RQ4.** Over 15 runs Xyntia always finds all 500 expressions in B1 and between 1051 and 1061 in B2. Thus, Xyntia is very stable across executions.

**RQ5.** Unlike Syntia, Xyntia performs very well on both B1 and B2 with a timeout of 60 s/expr. Fig. 4 reveals that it is still successful for a timeout of 1 s/expr. (78% proven equivalence rate), where it finds 2x more expressions than Syntia with a timeout of 600 s/expr.

We also observe such tendency over B1 and B1 (see Section 8.2) and for 12h timeout. On B1, Syntia reaches 41%, 74%, 88.2% and 97.6% success rate for respectively 1s, 60s, 600s and 12h timeout, against 100% success rate for Xyntia in 1s. For BP1, Syntia finds only 1/15 expressions with a 12h timeout against 12/15 for Xyntia in 60s. From evaluation on B1 and B2, it appears that Syntia success rate increases logarithmically over time. Thus, time budget needed for Syntia to catch Xyntia is expected to be unrealistic.

In addition, Xyntia handles well expressions using up to 5 arguments and all expression types (Table 4). Its mean quality is around 0.93, which is very good (objective is 1), and it rarely returns not equivalent expressions - only between 1.3% and 4.9%. Thus, Xyntia reaches high success and equivalence rate. It is fast, synthesizing most expressions in $\leq 1$ s, and it returns simple and correct results.

![Figure 4: Equivalence range of Xyntia and Xyntia$_{Xyntia$_{opt}$}$ depending on timeout (B2)](image-url)
We present in Fig. 5 the typical distance evolution along the search Xyntia is indeed guided by the distance function. Moreover, note

Table 4: Syntia & Xyntia (Xyntia\textsubscript{Opt} (HC), Random Walk (RW), Simulated Annealing (SA), Metropolis

- **Table 4**: Syntia & Xyntia (Xyntia\textsubscript{Opt} (HC), Random Walk (RW), Simulated Annealing (SA), Metropolis

<table>
<thead>
<tr>
<th></th>
<th>Bool.</th>
<th>Arith.</th>
<th>MBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ. Rate</td>
<td>53.8%</td>
<td>28.6%</td>
<td>21.1%</td>
</tr>
<tr>
<td>Xyntia</td>
<td>53.0%</td>
<td>27.8%</td>
<td>28.1%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.53</td>
<td>0.61</td>
<td>0.71</td>
</tr>
<tr>
<td>Succ. Rate</td>
<td>98.4%</td>
<td>96.5%</td>
<td>91.6%</td>
</tr>
<tr>
<td>Xyntia</td>
<td>97.8%</td>
<td>88.9%</td>
<td>94.9%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.73</td>
<td>1.0</td>
<td>1.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Expr.</th>
<th>MBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ. Rate</td>
<td>53.8%</td>
<td>28.6%</td>
</tr>
<tr>
<td>Xyntia</td>
<td>53.0%</td>
<td>27.8%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.53</td>
<td>0.61</td>
</tr>
<tr>
<td>Succ. Rate</td>
<td>98.4%</td>
<td>96.5%</td>
</tr>
<tr>
<td>Xyntia</td>
<td>97.8%</td>
<td>88.9%</td>
</tr>
<tr>
<td>Mean Qual.</td>
<td>0.73</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Title: 5.4 Optimal Xyntia and other S-Metaheuristics

## 5.4 Optimal Xyntia and other S-Metaheuristics

Previous experiments consider the Xyntia\textsubscript{Opt} configuration of Xyntia. It comes from a systematic evaluation of the design space. To do so, we considered (1) different S-metaheuristics: Hill Climbing (HC), Random Walk (RW), Simulated Annealing (SA), Metropolis Hasting (MH) and Iterated Local Search (ILS); (2) different sampling strategies; (3) different objective functions. This evaluation confirms that Xyntia\textsubscript{Opt} is locally optimal and that ILS, being able to restore best expression seen after a number of unsuccessful mutations, outperforms other S-metaheuristics (Table 5). Moreover, all S-metaheuristics – except Hill Climbing – outperforms Xyntia.

Table 5: Synthesis Equivalence Rate for different S-metaheuristics (B2, Xyntia\textsubscript{Opt}, timeout = 60 s)

<table>
<thead>
<tr>
<th></th>
<th>HC</th>
<th>ILS</th>
<th>SA</th>
<th>MH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equiv.</td>
<td>62.3 - 63.4%</td>
<td>90.6 - 94.2%</td>
<td>64.8 - 65.8%</td>
<td>57.7 - 58.5%</td>
</tr>
</tbody>
</table>

**Conclusion.** Principled and systematic evaluation of Xyntia design space leads to the locally optimal Xyntia\textsubscript{Opt} configuration. It notably shows that ILS outperforms other tested S-metaheuristics. Moreover, all these S-metaheuristics – except Hill Climbing – outperform MCTS, confirming that manipulating only terminal expressions is beneficial.

## 5.5 On the effectiveness of ILS over MCTS

We present in Fig. 5 the typical distance evolution along the search process when using Xyntia. We can see that the distance follows a step-wise progression, which is drastically different from the case of Syntia and enumerative MCTS (Fig. 3). Hence, unlike them, Xyntia is indeed guided by the distance function. Moreover, note that Xyntia globally follows a positive trend i.e., it does not unlearn previous work. Indeed, before each perturbation, the best expression found from now is restored. Thus, if iterative mutations follow a misleading path, the resulting solution is not kept and the best solution is reused to be perturbed. Keeping the current best solution is of first relevance as the search space is highly unstable and enables Xyntia to be more reliable and less dependant of randomness.

## 5.6 Limitations

Blackbox approaches must consider limited languages to be efficient. This restricts their use to local contexts – e.g., analyzing sets of code blocks rather than full modules.

Moreover, synthesis relies on two main steps, sampling and learning, which both show weaknesses. Indeed, Xyntia and Syntia randomly sample inputs to approximate the semantics of an expression. It then assumes that samples depict all behaviors of the code under analysis. If this assumption is invalid then the learning phase will miss some behaviors, returning partial results. As such, blackbox deobfuscation is unsuitable to handle point functions.

Learning can itself be impacted by other factors. For instance, *semantically complex expressions* are hard to infer. While they are rare in local code, we show in Section 8 how to take advantage of them to protect against blackbox attacks. A related problem are expressions with unexpected constant values. They are hard to handle as the grammar of Xyntia and Syntia only considers the constant value 1. Thus, finding expressions with constant values absent from the grammar requires to create them (e.g., encoding 3 as 1 + 1 + 1), which may be unlikely. A naive solution is to add to the grammar additional constant values but it significantly impacts efficiency. Indeed, for 100 values (\([0; 99]\)), the equivalence rate is divided by 2 (resp., by 4 for 200 values). Still, Section 7 shows that Xyntia can synthesize usual interesting constant values (unlike Syntia).

## 5.7 Conclusion

Because of the high instability of the search space, *Iterated Local Search* is much more appropriate than MCTS (and, to a lesser extent, than other S-metaheuristics) for blackbox code deobfuscation, as it manipulates terminal expressions only and is able to restore
We first compare Xyntia over the EA, VR-EA and EA-ED datasets. While STOKE integrates its own sampling strategy and grammar, CVC4 does not – thus, we consider for CVC4 the same sampling strategy as Xyntia (100 I/O samples with 5 constant vectors) as well as the Expr and Mba grammars. More precisely, CVC4Expr is used over B2 to compare to Xyntia (XyntiaOpt) and CVC4-Mba is evaluated on EA, VR-EA and EA-ED to compare against QSynth.

Our experiments show that CVC4-Expr and STOKE-synth synthesize less than 40% of B2 (respectively 36.8% and 38.0%) while Xyntia reaches 90.6% proven equivalence rate. Indeed enumerative search (CVC4) is less appropriate when time is limited. Results of STOKE-synth are also expected as its search space considers all assembly mnemonics. Moreover, Fig. 6a shows that blackbox and whitebox (STOKE-opti) synthesizers do not efficiently simplify obfuscated expressions. STOKE-opti finds only 1 / 500 expressions over EA-ED and does not handle jump instructions, inserted by the VM, failing to analyze VR-EA.

**Conclusion.** Xyntia rivals QSynth on light/mild protections and outperforms it on heavy protections, while pure whitebox approaches are far behind, showing the benefits of being independent from syntactic complexity. Also, Xyntia outperforms state-of-the-art program synthesizers showing that it is better suited to perform deobfuscation. These good results show that seeing deobfuscation as an optimization problem is fruitful.

## 7 DEOBFUSCATION WITH XYNTIA

We now prove that Xyntia is insensitive to common protections (opaque predicates) as well as to recent anti-analysis protections (Mba, covert channels, path explosion) and we confirm that blackbox methods can help reverse state-of-the-art virtualization [11, 34].
7.1 Effectiveness against usual protections

Xyntia is able to bypass many protections.

Mixed Boolean-Arithmetic [37] hides the original semantics of an expression both to humans and SMT solvers. However, the encoded expression remains equivalent to the original one. As such, the semantic complexity stays unchanged, and Xyntia should not be impacted. Launching Xyntia on B2 obfuscated with Tigress [11] Encode Arithmetic transformation (size of expression: x800) confirms that it has no impact: equivalence range with and without protection respectively equals 90.0 - 93.8% and 90.6 - 94.2%.

Opaque predicates [14] obfuscate control flow by creating artificial conditions in programs. The conditions are traditionally tautologies and dynamic runs of the code will follow a unique path. Thus, sampling is not affected and synthesis not impacted. We show it by launching Xyntia over B2 obfuscated with Tigress AddOpaque transformation (result: equiv. range equals 89.9 - 93.0%).

Path-based obfuscation [25, 35] takes advantage of path explosion to thwart symbolic execution, massively adding additional feasible paths. We show that it has no effect, by protecting B2 with a custom encoding inspired by [25] (result: equiv. range equals 89.5 - 93.7%).

Covert channels [31] hide information flow to static analyzers by rerouting data to invisible part of the states (usually OS related) before retrieving it – for example taking advantage of timing difference between a slow thread and a fast thread. Again, as blackbox deobfuscation focuses only on input-output relationships, covert channels should not disturb it. Note that the probabilistic nature of such obfuscations (obfuscated behaviours can differ from unobfuscated ones from time to time) could be a problem in case of high fault probabilities, but in order for the technique to be useful, fault probability must precisely remains low. We show it has no impact by obfuscating B2 with the InitEntropy and InitImplicitFlow (thread kind) transformations of Tigress [11] (result: equiv. range equals 89.0 - 94.0%).

Conclusion. State-of-the-art protections are not effective against blackbox deobfuscation. They prevent efficient reading of the code and tracing of data but blackbox methods directly execute it.

7.2 Virtualization-based obfuscation

We now use Xyntia to reverse code obfuscated with state-of-the-art virtualization. We obfuscate a program computing MBA operations with Tigress [11] and VMProtect [34] and our goal is to reverse the VM handlers.4 Using such a synthetic program enables to expose a wide variety of handlers.

Table 6: Xyntia and Syntia results over program obfuscated with Tigress [11] and VMProtect [34]

<table>
<thead>
<tr>
<th></th>
<th>Tigress (simple)</th>
<th>Tigress (hard)</th>
<th>VMProtect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary size</td>
<td>40KB</td>
<td>251KB</td>
<td>615KB</td>
</tr>
<tr>
<td># handlers</td>
<td>13</td>
<td>17</td>
<td>114</td>
</tr>
<tr>
<td># instructions per handlers</td>
<td>16</td>
<td>54</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xyntia</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completely retrieved</td>
<td>12/13</td>
<td>16/17</td>
<td>0/114</td>
</tr>
<tr>
<td>Partially retrieved</td>
<td>13/13</td>
<td>17/17</td>
<td>76/114</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Syntia</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completely retrieved</td>
<td>0/13</td>
<td>0/17</td>
<td>0/114</td>
</tr>
<tr>
<td>Partially retrieved</td>
<td>13/13</td>
<td>17/17</td>
<td>76/114</td>
</tr>
</tbody>
</table>

4Note that, as Syntia, Xyntia does not consider memory operations.

Tigress [11] is a source-to-source obfuscator. Our obfuscated program contains 13 handlers. Since at assembly level each handler ends with an indirect jump to the next handler to execute, we were able to extract the positions of handlers using execution traces. We then used the scripts from [7] to sample each handler. Xyntia synthesizes 12/13 handlers in less that 7s each. We can classify them in different categories: (1) arithmetic and Boolean (+, −, ×, ∧, ∨, ⊕); (2) stack (store and load); (3) control flow (goto and return); (4) calling convention (retrieve obfuscated function arguments). These results show that Xyntia can synthesize a wide variety of handlers. Interestingly, while these handlers contain many constant values (typically, offsets for context update), Xyntia can handle them as well. In particular, it infers the calling convention related handler, synthesizing constant values up to 28 (to access the 6th argument). Thus, even if Xyntia is inherently limited on constant values (see Section 5.6) it still handles them to a limited extent. Repeating the experiment by adding Encode Data and Encode Arithmetic to Virtualize yields similar results. Xyntia synthesizes all 17 exposed handlers but one, confirming that Xyntia handles combinations of protections. Finally, note that Syntia fails to synthesize handlers completely (not handling constant values). Still it infers arithmetic and Boolean handlers (without context updates).

VMProtect [34] is an assembly-to-assembly obfuscator. We use the latest premium version (v3.5.0). As each VM handler ends with a ret or an indirect jump, we easily extracted each distinct handler from execution traces. Our traces expose 114 distinct handlers containing on average 43 instructions (Table 6). VMProtect VM stack-based. To infer the semantics of each handler, we again used Blazytko’s scripts [7] in “memory mode” (i.e., forbidding registers to be seen as inputs or outputs). Our experiments show that each arithmetic and Boolean handlers (add, mul, nor, and) are replicated 11 times to fake a large number of distinct handlers. Moreover, we are also able to extract the semantics of some stack related handlers. In the end, we successfully infer the semantics of 44 arithmetic or Boolean handlers and 32 stack related handlers. Synthesis took at most 0.3 s per handler. Syntia gets equal results as Xyntia.

Conclusion. Xyntia synthesizes most Tigress VM handlers, (including interesting constant values) and extracts the semantics of VMProtect arithmetic and Boolean handlers. This shows that blackbox deobfuscation can be highly effective, making the need for efficient protections clear.

8 COUNTER BLACKBOX DEOBFUSCATION

We now study defense mechanisms against blackbox deobfuscation.

8.1 General methodology

We remind that blackbox methods require the reverser to locate a suitable reverse window delimiting the code of interest with its inputs and outputs. This can be done manually or automatically [7], still this is mandatory and not trivial. The defender could target this step, reusing standard obfuscation techniques.

Still there is a risk that the attacker finds the good windows. Hence we are looking for a more radical protection against blackbox attacks. We suppose that the reverse windows, inputs and outputs are correctly identified, and we seek to protected a given piece of code.
Note that adding extra fake inputs (not influencing the result) is easily circumvented in a blackbox setting by dynamically testing different values for each input and filtering inputs where no difference is observed.

**Protection rationale.** Even with correctly delimited windows, synthesis can still be thwarted. Recall that blackbox methods rely on 2 main steps (1) I/O sampling; (2) learning from samples, and both can be sabotaged.

- First, if the sampling phase is not performed properly, the learner could miss important behaviors of the code, returning incomplete or even misleading information;
- Second, if the expression under analysis is too complex, the learner will fail to map inputs to their outputs.

In both cases, no information is retrieved. Hence, the key to impede blackbox deobfuscation is to migrate from syntactic complexity to semantic complexity. We propose in Sections 8.2 and 8.3 two novel protections impeding the sampling and learning phases.

### 8.2 Semantically complex handlers

Blackbox approaches are sensitive to semantic complexity. As such, relying on a set of complex handlers is an effective strategy to thwart synthesis. These complex handlers can then be combined to recover standard operations. We propose a method to generate arbitrary complex handlers in terms of size and number of inputs.

**Complex semantic handlers.** Let $S$ be a set of expressions and $h, e_1, ..., e_{n-1}$ be $n$ expressions in $S$. Suppose that $(S, \star)$ is a group. Then $h$ can be encoded as $h = \prod_{i=0}^{n-1} h_i$, where for all $i$, with $0 \leq i < n$,

$$h_i = \begin{cases} h - e_i & \text{if } i = 0 \\ e_i - e_{i+1} & \text{if } 1 \leq i < n - 1 \\ e_{n-1} & \text{if } i = n - 1 \end{cases}$$

Note that $-e_i$ is the inverse element of $e_i$ in $(S, \star)$. Each $h_i$ is then a new handler that can be combined with others to express common operations – e.g., $x + y = h_0 + h_1 + h_2$ where $h_0 = (x + y) + -((a - x^2) - (xy))$, $h_1 = (a - x^2) - xy + -((y - (a \land x)) \times (y \land x))$ and $h_2 = (y - (a \land x)) \times (y \land x)$. Note that the choice of $e_1, ..., e_{n-1}$ is arbitrary. One can choose very complex expressions with as many arguments as wanted.

**Experimental design.** To evaluate our new encoding, we created 3 datasets – BP1, BP2 and BP3, listed by increasing order of complexity. Each dataset contains 15 handlers which can be combined to encode the $+, -, \times, \land$ and $\lor$ operators. Within a dataset, all handlers have the same number of inputs. Table 7 reports details on each dataset. The mean overhead column is an estimation of the complexity added to the code by averaging the number of operators needed to encode a single basic operator ($+, -, \times, \land, \lor$). Overheads in BP1 (21x), BP2 (39x) and even BP3 (258x) are reasonable compared to some syntactical obfuscations: encoding $x + y$ with MBA three times in Tigress yields a 800x overhead.

**Table 7: Protected datasets**

<table>
<thead>
<tr>
<th>#exprs</th>
<th>min size</th>
<th>max size</th>
<th>mean size</th>
<th>#inputs</th>
<th>mean overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP1</td>
<td>15</td>
<td>4</td>
<td>11</td>
<td>6.87</td>
<td>x21</td>
</tr>
<tr>
<td>BP2</td>
<td>15</td>
<td>8</td>
<td>21</td>
<td>12.87</td>
<td>x39</td>
</tr>
<tr>
<td>BP3</td>
<td>15</td>
<td>58</td>
<td>142</td>
<td>86.07</td>
<td>x258</td>
</tr>
</tbody>
</table>

**Evaluation.** Results (Fig. 7) show that Xyntia (with 1 h/expr.) manages well low complexity handlers (BP1: 13/15), but performance degrades quickly as complexity increases (BP2: 3/15, BP3: 1/15). Performances are similar with 1 s/expr. Syntia, CVC4 and STOKESynth find none with 1 h/expr., even on BP1. Actually, Syntia with 12 h/expr. gets only 1/15 success of BP1.

**Conclusion.** Semantically complex handlers are efficient against blackbox deobfuscation. While high complexity handlers come with a cost similar to strong MBA encodings, medium complexity handlers offer a strong protection at a reasonable cost.

**Discussion.** Our protection can be bypassed if the attacker focuses on the good combinations of handlers, rather than on the handlers themselves. To prevent it, complex handlers can be duplicated (as in VMProtect, see Section 7.2) to make patterns recognition more challenging.

![Figure 7: Xyntia (XyntiaOpt) on BP1, 2, 3 – varying timeouts](image)

**Figure 8: Example of a branch-less condition**

```c
// if (c == cst) then h1(a,b,c) else h2(a,b,c);
int32_t res = c - cst;
res = (((res ^ (res >> 31)) - (res >> 31)) >> 31) & 1;
return h1(a,b,c)*(1 - res) + res*h2(a,b,c);
```

**Datasets.** We introduce 5 datasets composed of 20 expressions. Expressions in dataset 1 are built with 1 if-then-else (ITE) exposing 2 basic handlers (among $+, -, \times, \land, \lor$); expressions in dataset 2 are built with 2 nested ITEs exposing 3 basic handlers, etc. Conditions are equality checks against consecutive constant values $(0, 1, 2, \text{etc.})$. For example, dataset 2 contains the expression:

$$ITE(z = 0, x + y, ITE(z = 1, x - y, x \times y))$$

5Available at : https://tinyurl.com/y34dcasus
Scenarios. Adding conditionals brings extra challenges (1) the grammar must be expressive enough to handle conditionals; (2) the sampling phase must be efficient enough to cover all possible behaviors. Thus, we consider different scenarios:

**Utopian** The synthesizer learns expressions over the MBA set of operators, extended with an \(\text{ITE}(\bullet = 0, \bullet, \bullet)\) operator (MBA + ITE operator set). Moreover, the sampling is done so that all branches are traversed the same number of time. This situation, favoring the attacker, will show that merged handlers are always efficient. **MBA + ITE** This situation is more realistic: the attacker does not know at first glance how to sample. However, its grammar fits perfectly the expressions to reverse. **MBA + Shifts** Here Xyntia does not sample inputs uniformly over the different behaviors, does not consider ITE operators, but allows shifts to represent branch-less conditions. **Default**. This is the default version of the synthesizer.

In all these scenarios, appropriate constant values are added to the grammar. For example, to synthesize Eq. (2), \(0\) and \(1\) are added.

![Figure 9: Merged handlers: Xyntia (timeout=60s)](image)

**Evaluation.** Fig. 9 presents Xyntia results on the 5 datasets. As expected, the Utopian scenario is where Xyntia does best. Still, it cannot cope with more than 3 nested ITEs. For realistic scenarios, Xyntia suffers even more. Results for Syntia, CVC4 and STOKESynth confirm this result (no solution found for \(\geq 2\) nested ITEs). Note that overhead here is minimal, and depends only on the number of merged handlers.

**Conclusion.** Merged handlers are extremely powerful against blackbox synthesis. Even in the ideal sampling scenario, blackbox methods cannot retrieve the semantics of expressions with more than 3 nested conditionals – while runtime overhead is minimal.

**Discussion.** Symbolic methods, like symbolic execution, are unhindered by these protections, for they track the succession of handlers and know which sub parts of merged handlers are executed. To handle this, our anti-blackbox protections can be combined with (lightweight) anti-symbolic protections (e.g. [25, 35]).

9 RELATED WORK

**Blackbox deobfuscation.** Blazytko et al.’s work [7] has already been thoroughly discussed. We complete their experimental evaluation, generalize and improve their approach: Xyntia with 1 s/expr finds twice more expressions than Syntia with 600 s/expr, some of which Syntia cannot find within 12h.

**White- and greybox deobfuscation.** Several recent works leverage whitebox symbolic methods for deobfuscation (“symbolic deobfuscation”) [5, 10, 22, 28, 30, 36]. Unfortunately, they are sensitive to code complexity as discussed in Section 7, and efficient countermeasures are now available [12, 25, 26, 37]– while Xyntia is immune to them (Section 7.1). Vidal et al. [16] recently proposed QSynth, a greybox deobfuscation method combining I/O relationship caching (blackbox) and incremental reasoning along the target expression (whitebox). Yet, QSynth is sensitive to massive syntactic obfuscation where Xyntia is not (cf. Section 6). Furthermore, QSynth works on a simple grammar. It is unclear whether its caching technique would scale to larger grammars like those of Xyntia and Syntia.

**Program synthesis.** Program synthesis aims at finding a function from a specification which can be given either formally, in natural language or as I/O relations – the case we are interested in here. There exist three main families of program synthesis methods [20]: enumerative, constraint solving and stochastic. Enumerative search does enumerate all programs starting from the simpler one, pruning snippets incoherent with the specification and returning the first code meeting the specification. We compare, in this paper, to one of such method – CVC4 [6], winner of the SyGus ’19 syntax-guided synthesis competition [2] – and showed that our approach is more appropriate to deobfuscation. Constraint solving methods [21] on the other hand encode the skeleton of the target program as a first order satisfiability problem and use an off-the-shelf SMT solver to infer an implementation meeting specification. However, it is less efficient than enumerative and stochastic methods [1]. Finally, stochastic methods [29] traverse the search space randomly in the hope of finding a program consistent with a specification. Contrary to them, we aim at solving the deobfuscation problem in a fully blackbox way (not relying on the obfuscated code, nor on an estimation of the result size).

10 CONCLUSION

Blackbox deobfuscation is a promising recent research area. The field has been barely explored yet and the pros and cons of such methods are still unclear. This article deepens the state of search-based blackbox deobfuscation in three different directions. First, we define a novel generic framework for search-based blackbox deobfuscation (encompassing prior works such as Syntia), we identify that the search space underlying code deobfuscation is too unstable for simulation-based methods, and advocate the use of S-metaheuristics. Second, we take advantage of our framework to carefully design Xyntia, a new search-based blackbox deobfuscator. Xyntia significantly outperforms Syntia in terms of success rate, while keeping its good properties – especially, Xyntia is completely immune to the most recent anti-analysis code obfuscation methods. Finally, we propose the two first protections tailored against search-based blackbox deobfuscation, completely preventing Xyntia and Syntia attacks for reasonable cost. We hope that these results will help better understand search-based deobfuscation, and lead to further progress in the field.